UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Chambers of Elrankat:
ARPG Game prototype
skill and item interaction system

Bachelor's thesis

Design and Development of Video Games

Surname: Pareja Piñol Name: Joan

Scheme: Pla d'estudis 2014

Director: Ripoll Tarré, Marc

Index

Abstract

The aim of this project is to create a game prototype for a diablo-like game. The focus of this prototype is in the skill and itemization system, it searches to create a skill system that allows the player to change the behaviour of the skills and its effects by equipping different items, as well as improving the player basic stats.

This project is focused on the programing of the said skill and itemization system using the unity API and C#, not on the art or design of the system or game.

Key Words

ARPG, Isometric, Videogame, Looting, Character development, skill system, items, Diablo-like

Table Index

Figure Index

Glossary

ARPG: Action role playing game

Isometric view: Angled viewpoint the scenery.

Asset: Any object or thing that forms part of the game.

Loot: Reward receive in games for completing objectives.

FPS: Frames per second

PoE: Path of exile

D3: Diablo 3

Stat: Value representing an the strength of a character in a field.

Skill: Ability for the character to deal damage

Buff: Object that modifies stats or skills

AoE: Area of effect

LMB: Left mouse button

RMB: Right mouse button

1. Introduction

1.1 Motivation

The creation of any kind of game is a very complex process that requires to put at use a wide range of skills, that's why it's usually done by teams.

Personally I have contributed to the creations of a few games in game jams or university projects, but I always focused my efforts in only one aspect of the game. Curiosity got the best of me and impulsed myself to do this project in order to have a better understanding and experience of all the parts that make make for a better game.

Also I have played a few ARPG games and dedicated a lot of hours in them, however even if I do enjoy the genre when searching for a new game of the genre to try new this I found myself paralysed looking at the amount of options that they tend to give to the new players, but at the same time when I passed this barrier I found myself having a fun time trying to get the most optimized build for my character. This is the reason that has driven me to try to create a ARPG game prototype that tries to have an easier entrance for new players offering the skills directly without having to worry about options but at the same time maintain the complexity in the loot and itemetitzation in order to preserve the fun of creating the most optimized builds.

1.2 Problem

Almost every ARPG in the market has very complex system regarding their character development by having skill trees that makes the player face with hundreds of choices every level they gain, or by having complex itemization systems that forces players to search for simulations to see which items would be better.

This complexity is appealing to the most hardcore fans of this genre however it often drives away newer players because of the amounts of information and choices that the player receives, resulting in a choice paralysis that makes the players stop playing the game due to not being able to decide which options may suit their playstyle.

Some games have approached this problem giving the players more predefined characters and predefined choices, but in this case the lack of choices has driven away veterans of the ARPG genre that enjoyed playing with a huge amount of options.

1.3 General goals

The objective of this project is to develop a prototype of an ARPG isometric game, putting special effort into the character development via skills and loot items while keeping the whole skill and loot system fairly simple for new players to the genre.

At the same time maintaining a slight level of complexity towards the itemization of the characters so more veteran players still find enough options to develop their own

builds, and also help new players get familiar with the typical amount of options that this games tend to offer.

1.4 Specific goals

Skill system: Allow to change the skills of their characters ingame in order to customize how their character is played, the skills should be able to be changed at any time to avoid a choice paralysis.

Characters: Allow players to drastically change their gameplay without breaking a character appearance and lore.

Equipment: Items will be a pillar of this project, to create a good looting system we need first to implement a way of creating items with random values that modify the stats of the player, and even to a more deeper level modify the skills themselves, giving the player a way of

Enemy AI and combat: Even though the AI in this games is rather simple and consists mainly of enemies blindly going towards the players and attacking them, we need to have a little bit of diversity like ranged enemies and melee enemies to give players a few options in how to beat the enemies.

Level Design: Design levels to guide the players during the first steps he takes in this game and teach them how to progress in the game.

1.5 Project scope

The target of this project primarily focused towards new players that want to try the ARPG genre, they are usually scared to enter this type of game due to the huge amount of information that they receive initially, and also during their character development they keep receiving tons of information that makes them quit the game or they ignore and have a difficult time afterwards playing the game.

However this project is also focused towards other ARPG games, since it's trying to explore new options to preserve the complexity of this games while offering a simpler and less intimidating look for new players. In that regard other ARPG could take ideas from features implemented in this prototype and further develop them to implement them in their games, lowering the ARPG entry barrier attracting more players to their games.

2. State of art

2.1 Game Engines

Nowadays to create a game is essential to use a game engine, games are becoming more complex as time passes and it's impossible to manage all of the aspects of a game without a proper game engine.

Big companies usually create and use their own game engines to create their games since it offers the advantages of not paying licences for external game engines and allows them to modify the game engine itself to adapt it to the games that the company usually produce, making it easier and faster for them to create their games.

However creating a game engine its not an easy task and requires a lot of time and knowledge, which translate into more money and production time invested by companies or individuals, for this reason projects with a small team behind them tend to use one of the engines already available at the market.

There are a lot of game engines available both free and licensed, most of the engines though tend to focus in either 2D or 3D development, since this project aims to develop a 3D game using some external assets to speed up parts of the development we will focus in Unity and Unreal since both are focused for 3D development and have a dedicated shop for assets.

2.1.1 Unity

Unity is without a doubt the most accessible engine at the market, Unity has been developed with the idea of democratizing game development, and so it has become one of the most user-friendly engines in the market having a really low entrance barrier for new users. It also is one of the best generic engines, meaning that it doesn't focus in any type of game and instead offers a wide range of tools to create game of any kind and for almost any platform.

Unity also provides an online asset store where users can upload their creations or unity plugins to sell or give them away to other users. Which thanks to the "Game democratization" ideology that Unity has been built upon a lot of users have been encouraged to create and upload their tools or assets over the years, converting the asset store into a huge source of different assets and tools which help in the creation and development of new games.

In terms of programming Unity offers its own scripting API using C# language, the API itself is really complete and comes with an extensive documentation on how to use it, it also offers a lot of options to customize or create a new API for custom tools that one builds in the engine. However since its build to be easily accessible and easy to understand for new developers it is hard to access some advanced features and makes it harder to optimize parts of the code such as managing memory.

Unity will be used to develop this project thanks to both the easy to use interface and the benefit of having a huge asset store. Developing a game prototype is already a complex task by itself so having an easy to use interface and a well documented API will reduce time on learning how to use the engine and allow more time for the development, the same way the asset store can save us time by providing already made assets, such as art, ready for us to use and will save us time. Also since the objective is to make a game prototype it is not necessary to have the best optimization possible, and as long as the game runs smoothly the majority of the time a few fps drops can be allowed.

2.1.2 Unreal

Unreal engine is another easily accessible engine in the market created by Epic games, often seen as a more professional choice of the free to access engines in the market, it does not have a direct license fee and instead is offered for free in exchange of 5% of your game/app revenue.

Unreal handles programing by either using its own C++ visual programing tool, called blueprint, or by directly writing C++ code. Thanks to it Unreal allows a much better optimization of the games created using its engine compared to other engines such as Unity, which in turn allows for a better graphic quality. Also unreal offers their source code directly meaning that one can add a missing feature in case that they need it, or patch some critical error in a brute way to make their game work.

However Unreal also has a very high entry barrier, and a not really good user experience. Apart from the typical barrier that any engine has that makes the user understand the different objects of the scene and how they link to each other and work the Unreal blueprint system is quite complex and requires a lot of time to fully understand it. One major flaw of Unreal is that the game and the engine are executed together, meaning that if for some error the game crashes or freezes Unreal engine will also do the same, which will slow down the process of debugging and finding critical errors.

Unreal as also developed an online asset store called “Unreal Marketplace” which allows users to upload and sell their creations to other game developers. One of the best things about this stored compared to other asset stores is the fact that every month gives a few asset packages for free, and also it releases assets for free from previous canceled games or projects that Epic games has made in the past such as Paragon.

It has been decided not to use Unreal mainly because it has a higher entry barrier compared to other engines, the project it’s already complex by itself and it wouldn’t be wise to spent more time in the development because it’s spent on learning how to use the tool. Also the main benefits of using unreal will be a better optimization and graphic quality which will not benefit much the project since being a prototype a low graphics and a few fps drops are acceptable.

2.2 ARPG Games

Creating a game means that we will face an enormous competition from all other games that are already at the market or will be in the near future, however not all players like the same type of game so we shall look at the current games that fit our genre, in this case the isometric ARPG genre.

In this type of genre the majority of games give a lot of importance to the looting system, although every game has his own peculiarities the gameplay core is found in character development via the different skill that can be acquired and how to power up this character using the differents items that the player founds, typically as random loot for monsters or rewards for completing objectives.

This development usually puts a lot of weigh in the loot obtained, each item or piece of equipment found gets random generated values that boost the character stats such as speed, health, damage, ...etc. Thanks to this random generated loot it's very rare to find two exact copies of the same item, since if you can find multiple copies of the same sword that give different boosts to the character and so you may want to keep both copies to use in different situations.

In some cases items can even have slots or sockets, which are generally used to insert gems in it and getting a different buff depending on which type of gem is used, giving to the player a wider range to customize how her character will work, and at the same time expanding the different variations of the same item.

2.2.1 Path of Exile

Even if it this game has been out for a while the developers have adopted a game as a service model and continue to deliver new updates to the game, and thanks to the last updates and the game being free to play Path of Exile (PoE) has received an incredible boost in popularity in the last years.

One of the most noticeable characteristic of this game is that their system of skills and character development are quite different than the typical game of this genre.

In the typical ARPG each character has a skill tree which displays all the available skills for the character, and as the character levels up the skill can be unlocked, and then the player gets equipment that boost either the skills or the basic stats of the character. In Path of Exile however leveling up gives an option to the player to level up the character stats, and the skill must be obtained by equipping skill gems (obtained as loot) to a specific socket in their equipment.

Even more, the sockets on the equipment have an specific order and color and can be linked in various ways, which allows the player to insert support gems which will boost the effect of other skill gems depending in the color of the socket.

Figure 1. PoE Item sockets

Regarding the leveling skill tree path of exile has an incredible huge skill tree that displays at the same time all the possible points for every character class in the game, and the starting point on that huge tree will be determined by the character class. This is a clear example how the huge amount of possibilities may paralyse a new player and lose their interest in the game.

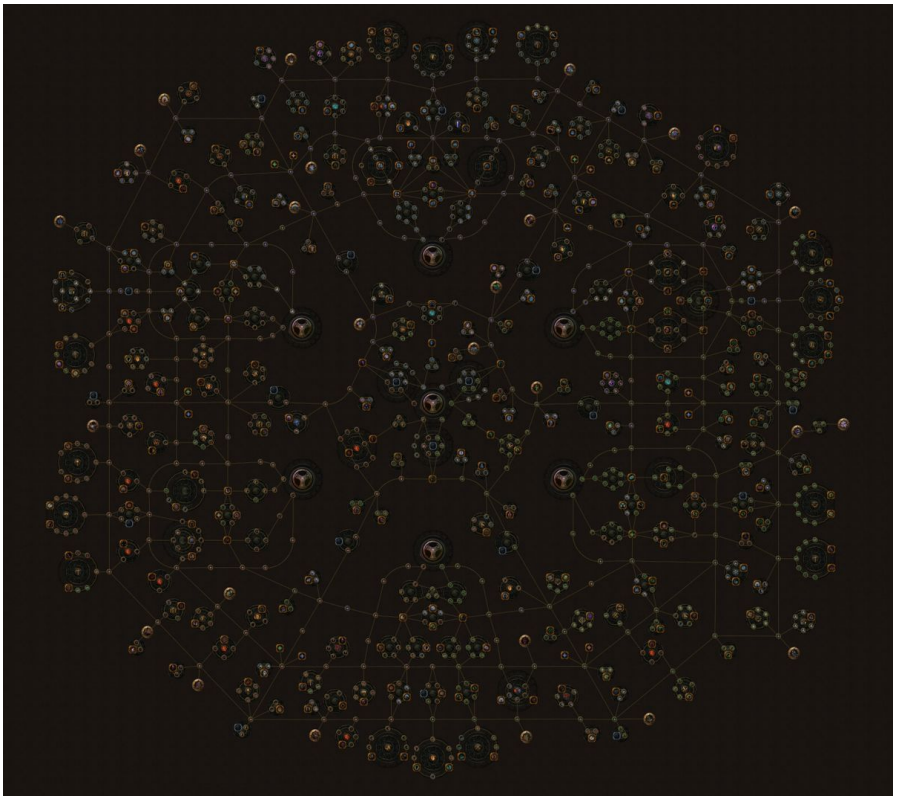


Figure 2. PoE skill tree

2.2.2 Diablo 3: Reaper of Souls

Diablo 3 (D3) has changed quite a bit over the years since it's first day, even more with the addition of the expansion Reaper of Souls, in this part it will be analysed the current state of Diablo 3 with its expansion.

D3 skill system is quite simple compared to other ARPGs or its predecessors, in D3 the skills are simply listed in a menu and are unlocked passively when the character levels up, also along with the skill leveling up a character will unlock different runes for each skill, this runes modify the skill usually changing its behaviour and the damage it deals. Once the skills or runes are unlocked the player can equip up to 6 different skills, each with their rune variation, and can freely swap between them. This approach solve the choice paralysis problem that other games have with their huge skill trees, however it also creates another problem, since when the player has obtained the skills which it wants to use, there is no reward or motivation to level up the character.

This lack of motivation for the level up is further amplified when we look at the D3 looting system, the items the player receives have a level that is based on the current level of the character, and the most powerful one won't start to appear until the player is close or has achieved the maximum level, in this case 70. This is a big problem since if the player gets the skills it wants in early levels it will find a huge gap of no rewards until their character reaches level 70, and it may make the player quit the game.

Looking at the itemization we can see that the great majority of items found by the player just boost the characters basic stats, but the most rare items (Legendary and Set) apart from boosting the character stats offer a unique attribute that can either insanely boost a skill even modifying their behaviour, or they can also give passive skills to the character. There's a special type of legendary items called set items that are grouped into different sets, this items don't give any special boost by themselves but equipping various parts from the same set will give a unique boost or skill like legendary items, usually much larger and important since they require more equipment slots.

Set items create another important problem that should be avoided, the problem itself it's not found on the set mechanic itself but rather in its benefits to the player, since Set items occupy various equipment slots to get give their boosts the boosts on the set items are really strong, so strong that overshadow all other legendary combinations that take their spot. This is a problem since there are some players that enjoy to spent time combining different legendary items and its effects to create their own custom builds, and even though it can still be done those builds can't compete with the plain set boost that is given by the developers, meaning that those player may quit the game.

3. Project planning

3.1 Planning and tracking tools

3.1.1 GANTT and Trello

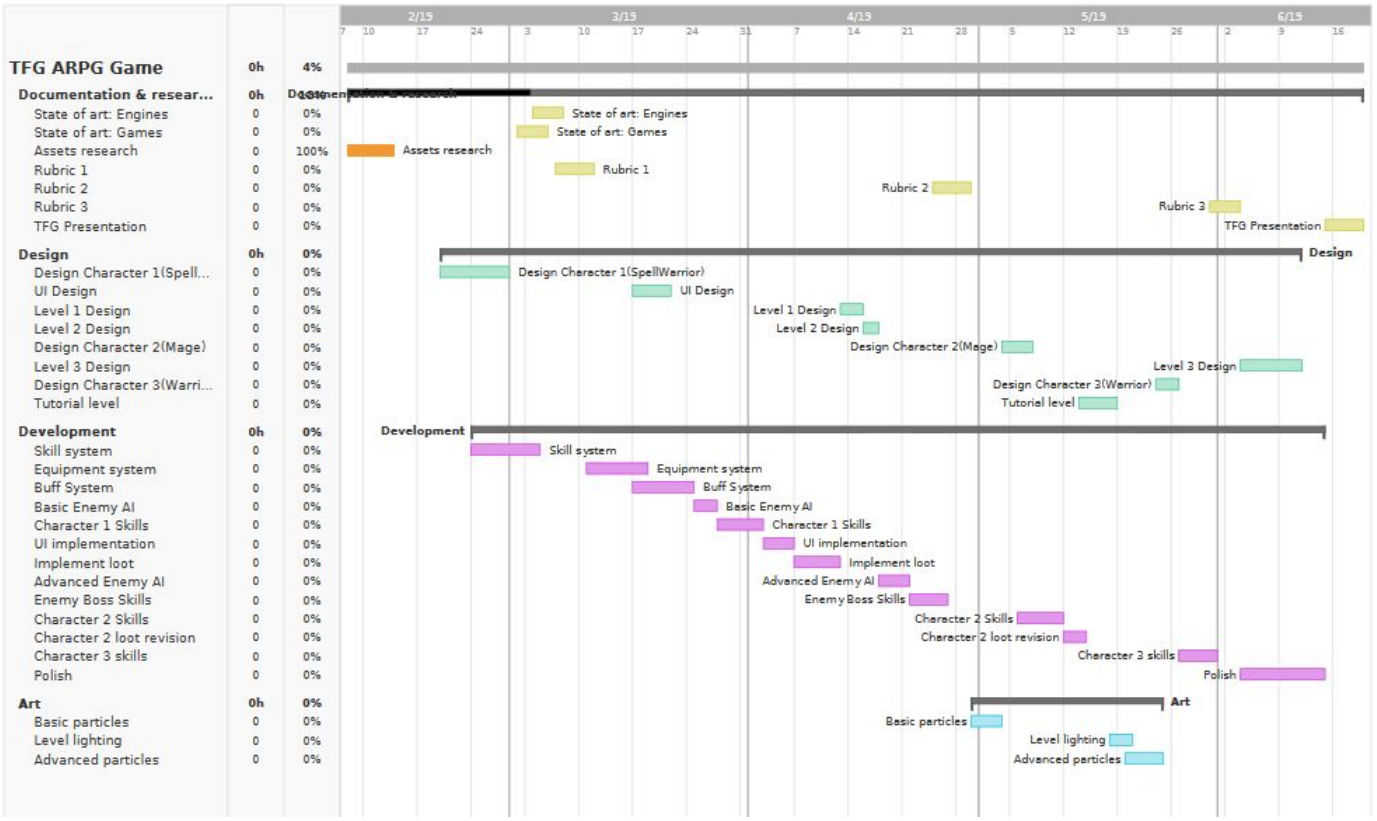


Figure 3. GANTT

A Gantt chart will be used to keep a general track of the taks the project requires, Trello will also be used during sprints to help visualize the workload for smaller periods of time, since Trello will be used during sprints to complete the major tasks in the Gantt it will allow for a subdivision of these task for easier management.

3.1.1.2 New planning

The initial planning was redone shortly after the delivery of the first rubric, since after a few days of working on the project it was clear that the time for developing tasks was estimated to be too short, and some task were unrealistic to complete in time. For this reason another schedule was made to focus more in the prototype itself, removing some tasks the may shift the focus of the prototype and also adjusting the time of development to better suit this prototype. One important change was to develop the item, buffs, and skill all together since they were too related with each other and would have been too difficult to develop independently.

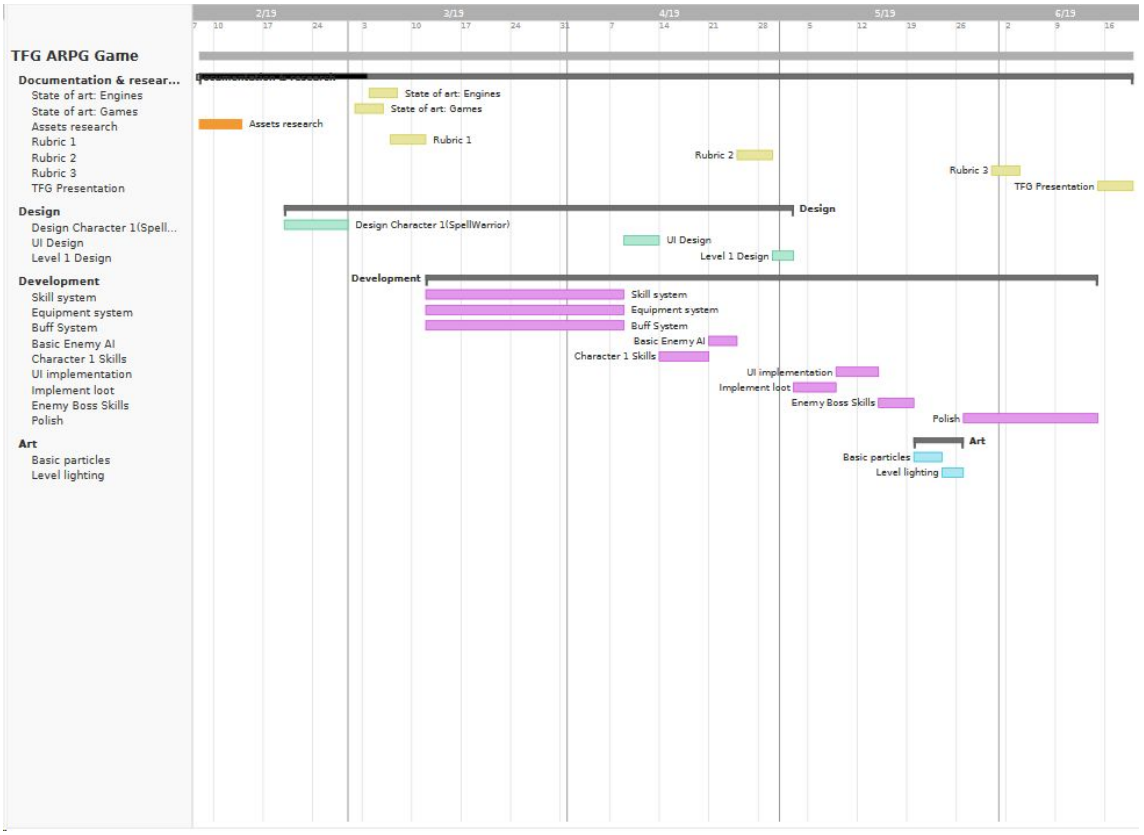


Figure 4. New GANTT

3.1.2 GitHub

Github will be used to keep track of the development and also use it as a platform de deploy the different releases that the project may have, it will also help to provide an online backup for our files.

However due to the use of external assets keeping all the projects files in a public repository will be forbidden, so only the part of the project that consists of own assets and code will be uploaded to github.

3.2 Validation tools

When validating the project two things will be taken into consideration, on one hand that the build is stable and on the other hand that the build is fun to play.

To validate the stability of the build internal playtesting sessions will be organized for each new feature added with to objective to approach these features and try to "break" the game in any way possible in order to find possible crashes, bugs or exploits.

In order to validate if the game is fun to play it will be required to create an external playtesting session using individuals not involved in the development of the game in order to prevent a biased opinion of the game. Those sessions will require a previous

questionnaire to the participants to know her background which will allow a better understanding of their decisions during the playtesting session.

3.3. SWOT

STRENGTHS	WEAKNESSES
● Having a simpler game may attract new players. ● Not searching economic benefits allows to focus more on the gameplay. ● This genre is easily expandable	● Lack of artistic skills. ● Lack of experience developing this games, may lead to planning errors.
OPPORTUNITIES	**THREATS**
● Currently there's a void for the best ARPG in the market and drives players to search for new ARPGs which will us a small boost in visibility. ● Fanbase in ARPG tends to be very loyal and maintain interest for a long time.	● Some AAA companies are working on ARPG games and may have already solve the problem. ● Keeping the game too simple may bore veteran ARPG players. ● The ARPG genre has a nich audience.

Table 1. SWOT

3.4. Risks and contingency plans

Lack of time:

As said in the SWOT the lack of experience in developing this kind of games may lead to errors in the planning resulting in a lack of time to completely develop the project. To minimize this effect we will work with a feature driven development. This genre of games has a core mechanic than then is expanded to add depth to the game, and so the development will focused in finishing the core mechanic as soon as possible so we can discard some of the last tasks and still have a closed game.

Lack of artistic skills:

Even if the prototype is focused in the gameplay aspect it still need to have a minimal degree of art in order to feel appealing to the public. In order to help minimize this weakness we will get the basic art for the world from the Unity asset store and the

only artistic focus of this development will be the particles. In case that it's impossible to achieve good enough particles they will also be taken from the asset store.

3.5. Cost analysis

This project will be developed by only one person however the cost analysis has split the single developer into the different roles that will be taken.

	FEB	MAR	APR	MAY	JUN
Costs					
Programer	-500,00 €	-500,00 €	-500,00 €	-500,00 €	-500,00 €
Designer	-500,00 €	-500,00 €	-500,00 €	-500,00 €	-500,00 €
Artist	-500,00 €	-500,00 €	-500,00 €	-500,00 €	-500,00 €
Office	-350,00 €	-350,00 €	-350,00 €	-350,00 €	-350,00 €
Google drive	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
Unity 3D	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
Assets provider	-120,00 €	0,00 €	0,00 €	0,00 €	0,00 €
Hardware depreciation	-46,17 €	-46,17 €	-46,17 €	-46,17 €	-46,17 €
Total	-2.016,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €
Cummulative Total	-2.016,17 €	-3.912,33 €	-5.808,50 €	-7.704,67 €	-9.600,83 €

Table 2. Cost analysis

Assumptions: Office costs will only account for gas, water and electric bill since the developer will work from home without the need to pay rent.

	Asset value	Amortization period (years)	Months	Annual depreciation	Monthly depreciation
Computer	2.000,00 €	4	48	500,00 €	41,67 €
Screen	200,00 €	5	60	40,00 €	3,33 €
Keyboard and mouse	70,00 €	5	60	14,00 €	1,17 €
TOTAL	2.270,00 €			554,00 €	46,17 €

Table 3. Hardware depreciation

The project consists of making a game prototype and there are no plans of selling any content created, however an estimation plan has been made expanding the production time to complete the game. The estimation has been made for a regular, best and worst case scenarios.

	FEB	MAR	APR	MAY	JUN	AUG	SEP	OCT	NOV	DEC
Income	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	262.325,00 €	52.465,00 €	12.591,60 €
Cummulative	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	262.325,00 €	314.790,00 €	327.381,60 €
Costs	-2.141,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €
Cummulative	-2.141,17 €	-4.162,33 €	-6.183,50 €	-8.204,67 €	-10.225,83 €	-12.247,00 €	-14.268,17 €	-16.289,33 €	-18.310,50 €	-20.331,67 €
Balance	-2.141,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	-2.021,17 €	260.303,83 €	50.443,83 €	10.570,43 €
Cummulative	-2.141,17 €	-4.162,33 €	-6.183,50 €	-8.204,67 €	-10.225,83 €	-12.247,00 €	-14.268,17 €	246.035,67 €	296.479,50 €	307.049,93 €

Table 4. Best case scenario

	FEB	MAR	APR	MAY	JUN	AUG	SEP	OCT	NOV	DEC
Income	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	31.479,00 €	10.493,00 €	1.049,30 €
Cummulative	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	31.479,00 €	41.972,00 €	43.021,30 €
Costs	-2.016,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €
Cummulative	-2.016,17 €	-3.912,33 €	-5.808,50 €	-7.704,67 €	-9.600,83 €	-11.497,00 €	-13.393,17 €	-15.289,33 €	-17.185,50 €	-19.081,67 €
Balance	-2.016,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	29.582,83 €	8.596,83 €	-846,87 €
Cummulative	-2.016,17 €	-3.912,33 €	-5.808,50 €	-7.704,67 €	-9.600,83 €	-11.497,00 €	-13.393,17 €	16.189,67 €	24.786,50 €	23.939,63 €

Table 5. Regular case scenario

	FEB	MAR	APR	MAY	JUN	AUG	SEP	OCT	NOV	DEC
Income	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	1.049,30 €	209,86 €	0,00 €
Cummulative	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €	1.049,30 €	1.259,16 €	1.259,16 €
Costs	-2.016,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €
Cummulative	-2.016,17 €	-3.912,33 €	-5.808,50 €	-7.704,67 €	-9.600,83 €	-11.497,00 €	-13.393,17 €	-15.289,33 €	-17.185,50 €	-19.081,67 €
Balance	-2.016,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-1.896,17 €	-846,87 €	-1.686,31 €	-1.896,17 €
Cummulative	-2.016,17 €	-3.912,33 €	-5.808,50 €	-7.704,67 €	-9.600,83 €	-11.497,00 €	-13.393,17 €	-14.240,03 €	-15.926,34 €	-17.822,51 €

Table 6. Worst case scenario

4. Methodology

For the development of this project it will be used the agile methodology of Feature-driven development (FDD), supported with the scrum methodology.

FDD consists of analysing the overall project and divide the project in different “Features”, each feature consists of an independent part of the project that will add functionality to the game. Once those features are defined they will be organized and ordered depending on the priority they have in order to complete the project, they will also be given a specific time to be completed.

For each feature conceived in the general planning a small plan will be made to subdivide the feature in smaller task that will later be completed, here is where scrum will come in handy since each feature will correspond to a sprint, at the beginning of the feature development period all the smaller task will be put in a backlog, during the development period the task will be review daily to know if there is an error in the estimated time and try to solve it as soon as possible. At the end of each feature sprint all the tasks should be closed, in case that some tasks are not closed we will analyse it and see how it can affect the overall project, and depending on the result of this analysis the task will be discarded or given more time to complete at the expense of discarding other less important features.

5. Development

Before starting the development of the prototype it's necessary to define the general idea of the game we want to prototype, and what's most important which parts of the game will be the focus of this prototype.

In this case the aim of the project was to develop a prototype for an ARPG game with an isometric view, focusing the prototype towards the itemization and character development systems that this type of games offer.

5.1. Design of the prototype

The most important thing to keep in mind was how the player will interact with our game, in this case the most important interaction in this prototype will be in how the player is able to upgrade their character by acquiring new items, so the design will be centred in how the items or skills are able to apply different buffs modifying the character stats or skills.

However we still need to at least roughly design the most basic features of the game, which would be how the player moves, attacks, obtains items... . Also we need to to define which stats the character will have in order to compute the damage dealt, damage taken, movement speed, and other kinds of game elements that can be modified by those stats.

5.1.1. Character stats

Before entering into further detail on how the player can use the skills it's important to define which stats will be used in order to compute the various actions the player can make and also how will the various buffs from items or skills affect the player.

Stat name	Explanation
Max health	Maximum health of the character.
Health regen.	Amount of health regenerated every second.
Max resource	Maximum resource of the character, used to cast skills.
Resource regen.	Amount of resource regenerated every second.
Move Speed	Amount of units the player move with each step.
Weapon damage	Base value to compute damage dealt.

Attack speed	Times the player can attack each second.
Armor	Reduces all incoming damage by a set amount.
Physic res.	Reduces physical damage received by a set amount.
Fire res.	Reduces fire damage received by a set amount.
Water res.	Reduces water damage received by a set amount.
Shock res.	Reduces shock damage received by a set amount.
Earth res.	Reduces earth damage received by a set amount.

Table 7. Character stats

Player stats will be the main focus for the character progression in this game, player skills, damage dealt, movement, damage received…, will be affected by the values of the player stats, and so character progression will be linked on how the player is able to affect those stats using various skills or buffs, which will be explained on more detail later on.

With this in mind it's important to define how the stats values can increase or decrease. In this game those values will be able to increase either in a multiplicative way or an additive way, it's important to control how those increases are applied since it's easy to make one increase irrelevant depending on the order they are are applied, in this game it will be computed the following way:

$$total\ value = (base\ value + buff\ additions) * buff\ multiplications$$

This computation prevents additive values from being irrelevant when the multiplicative values are big, since they will also be multiplied and thus bigger, which will make them prevail relevant. Similarly with multiplicative values it's sure that they will always multiply the biggest possible value, and not be rendered useless by multiplying a small base value while it's buffed by bigger addition values.

5.1.2. Movement and damage computation

For this game the player will move the same way as most typical ARPG, by using their left mouse click to select the destination point of the character and their character will move the destination.

Regarding the attack the player will be able to use a basic attack and up to 5 extra skills that the player can assign to a skill bar, while each skill will activate differently depending on what it does, some may shot a projectile towards the cursor, others may just damage an area around the player, etc… the basic attack will be linked to the left mouse click, the same way the movement works, when the player presses the mouse left key depending if there's an enemy or not in the cursor the player will either attack

or move towards the destination, if there's an enemy they will attack it, and if there is no enemy they will move to the point.

For the damage computation the objective was to give the player various ways of defending against incoming damage, to achieve this the damage was given a type (Physical, Fire, Water, Shock or Earth) and when an actor, being a player or an enemy, receives damage the damage value is reduced first by armor (which serves as general resistance for every type), and then is reduced again by the resistance of the same type as the damage, meaning that a fire attack will be reduced by armor and then by the fire resistance of the actor receiving the attack.

Another consideration when dealing with damage was to find which formula was needed to compute the amount of reduced damage, the objective was that at lowers level of armor or resistance the damage reduction was quite important so the players feels motivated to increase their resistance but at the same time lower the impact of damage reduction at higher values of resistances. One last thing to consider was that the player shouldn't be able to get a 100% of damage resistance.

One formula that fulfilled all those requirements was the following one, being X the current amount of armor or resistance and Z the amount of armor or resistance needed for a 50% damage reduction.

$$dmg\ reduction = \frac{Z}{Z+X}$$

Figure 5: Damage formula (scale of 1)

5.1.3. Skills, Items and buffs

The main focus of this prototype is the interaction present between the items, skills, and buffs, and so it was needed to establish a decision on how they will interact.

Skills are the most important of the 3, since both items and buffs may influence the skill either directly or indirectly.
Skills are the main tool the player has to defeat their enemies, and also are an important tool for the player to customize their playstyle and even gain a feeling of progression.

Every game has their own way of dealing with their skill system, either using complex skill trees or a simple selection of available skills, for this game it was opted to go for a simple selection of available skills since one of the objective of this game is to avoid the overwhelming feeling that the players can have when presented with huge skill trees.

Figure 6: Ingame skill selection, equipped skills in yellow

The other 2 main pillars of the game, items and buffs, are quite linked.
Items are in essence carriers of buffs, each item has also a class which determines where the player must equip the item, for example an item from the hemlet class can only be equipped in the head slot. This is important since the player can only equip one item on each slot, and it needs to be equipped for the buffs to affect the player.

This link is what helps create one of the interesting aspects of this game, which is the players searching through their inventory to find the most optimal items to equip. For that purpose the players need to know what every item does, since some items can have quite a lot of buffs the buffs and information about the item will only be shown when the player hovers the mouse over them.

Figure 7: Ingame inventory, displaying info of a helmet.

Buffs are simpler to describe, in short their main purpose is to modify the player basic stats or the behaviour of skills.
In a more detailed view most of the buffs will only increase the base stats of the players, either by multiplying them or adding a flat value as described before in the stats section. However there is also special buffs that instead of focusing on increasing

a stat they modify a skill, either by increasing the skill damage, range or other attributes it can have, or by directly modifying how the skill behaves ingame.

5.1.3.1. Skills design

Here are described the various skills designed for this game with a short explanation of what they are supposed to do:

ICON	NAME	DESCRIPTION
	Arcane Explosion	Projectile, when hitting an enemy produce an AoE explosion
	Dash	Dash to one direction damaging enemies.
	Dragon breath	Damages enemies in a cone.
	Earthquake	Stuns and deals damage in a circle and leaves a slowing area for 5 seconds.
	Ice armor	AoE sphere around the player, follows the player. Increases armor by 100 and damages enemies inside it.
	Shield bash	Throw enemies in front towards the direction of the skill, stunning them.

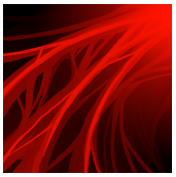
	Spirit refuge	AoE zone in a circle, heals player during its duration.
	Sting	Heavily damages the first enemy hit in a straight line.
	Swing	Damages all enemies around the player.
	Thundercall	Deals damage in a sphere AoE.
	Wind Fury	Increase attack speed for 30 seconds.
	Wind Rush	Projectile, damages and knockback enemies.

Table 8: Skills icons, names, and description

5.1.4. Level Design

The focus of the design for the level in this prototype was to make a level that had an abundance of big areas that can be filled with enemies, since defeating them is one of the main game loops that this type of games have. Another aspect that was searched for the design of the level was that the level itself resembles a labyrinth, having multiple paths for the player to explore.

The image below is the final map of the level, marking with pink the areas where enemies will be, and the start marked with a cyan line. This level has quite a few rooms to fill with enemies and also has this labyrinth feel, the first parts of the level however were left more linear in order to not confuse the player during the beginning.

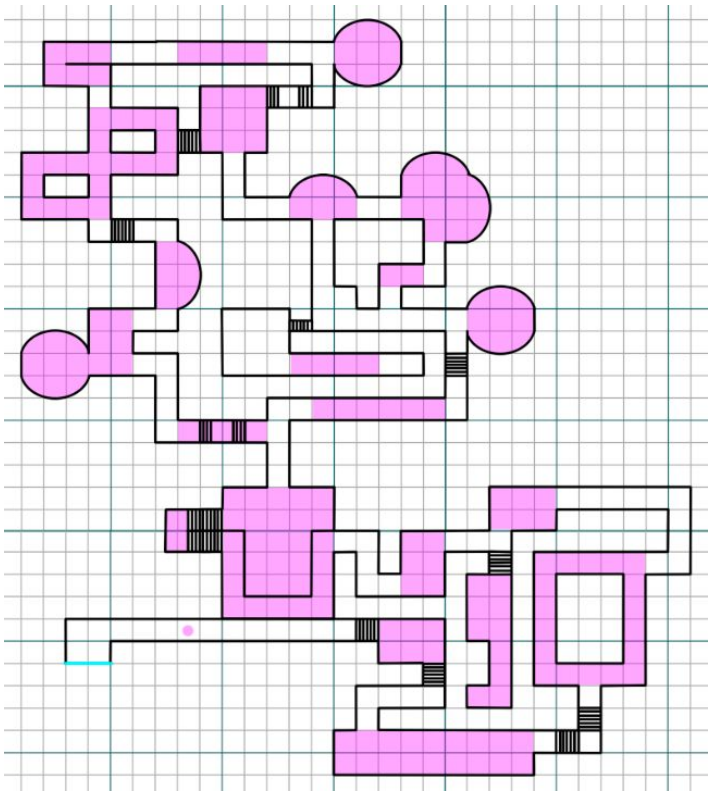


Figure 8: Level map, pink areas represent enemy filled zones.

5.2. Implementation

5.2.1. Input reading

For this prototype as well as for most of the diablo-like ARPG games the input of the player completely relies on the mouse position, one way of translating the mouse position in the screen to an ingame position is the use of raycast, which is the technique used for this prototype.

A raycast consists on creating a line (ray) between 2 points in the worldspace and then check all the objects we have to see if they collide or not with the ray. Checking all the objects of the scene is as it seems a tedious task, engines like unity have their own methods to help reduce the cost of this task, one of this optimizations is accessible to the users and its to limit the search to specific layers, layers are the system that unity uses to sort the objects in the scene into different groups.

For this prototype it was only needed to translate the mouse position in certain cases, one, when the mouse is over terrain so the user can move towards the position, two, when the mouse is over an enemy so the player can attack it or follow his movement, and three, when the mouse is over an item so the player can pick it up. Knowing this it's possible to make the raycast faster by only checking this 3 layers (enemies, terrain, items) that the player can interact with.

Since the prototype is developed in Unity it's quite easy to create the raycast, Unity itself provides a method to create a ray inside its camera component, so it's possible to crete this raycast by using the Camera.ScreenPointToRay() method to get a ray, and then create the raycast by the usual Physics.Raycast() method.

Using these methods we will have the exact point where the raycast lands, for this prototype however it's not interesting to have the exact point where the raycast lands, but is better to have an flattened approximation. This will help when calculating for example the direction of a skill projectile, since the exact hit point may be the feet of the enemy the projectile may go downwards and look weird, if instead the raycast hit point is normalized to a plane it will towards the enemy in a straight line. This process is only need when dealing with enemies, and it's achieved by replacing the hit point with the enemy position and shifting so it's on the head of the enemy, for this prototype the shift values was (0, 1.5, 0).

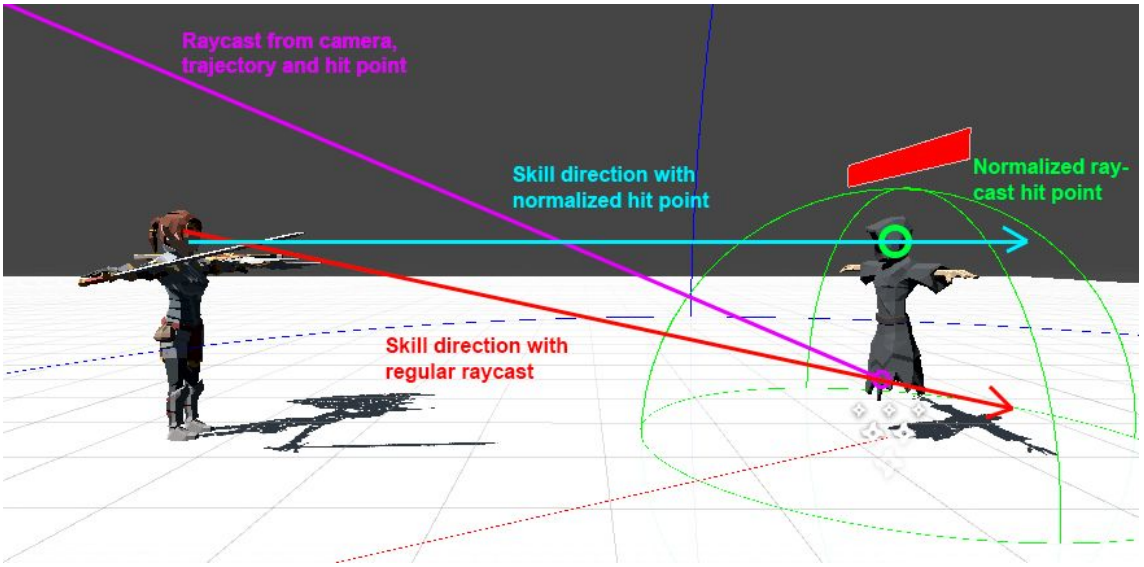


Figure 9: Raycast normalized explanation

With the raycast info acquired and normalized in the plane it's easy now to send it to the different controllers so the ingame character responds to the player inputs.

5.2.2. Controllers

Controllers take care of organizing how all the different elements of the game affect the player character, for the sake of code organization the functionality has been divided into 3 different controllers, character controller, movement controller and skill controller, which are explained below.

5.2.2.1. Character Controller

The Character controller is a general controller for the character and has all the base data for the character and controls how this data is used. Since this controller need to be easily accessible to read the basic data of the character it is implemented as a singleton by having a pointer to itself in the class definition, which is a simple way of creating a singleton in unity.

```
public static CharacterController instance;
private void Awake()
{
    if (instance == null)
        instance = this;
    else
        Debug.LogError("More than one character controller detected");
}
```

Figure 10: Simple singleton in unity

Since the character controller is a singleton is the best place to store the stats of the player, so we have a place to get all the player stats references when we need to call them, since the character controller has all the character stats it's also the one in charge of computing the damage that the player receives, applying both armor and resistance mitigations using the damage mitigation formula designed.

```
float armor_mitigation = dmg_half_reduction / (dmg_half_reduction + variables_stats[(int)StatId.Armor].Buffed_value);
float res_mitigation = dmg_half_reduction / (dmg_half_reduction + variables_stats[(int)type].Buffed_value);

true_damage = dmg_value * armor_mitigation * res_mitigation;
```

Figure 11: Damage formula code

The character controller is also the script that reads all the player inputs and takes the necessary actions, calling other controllers or sending events so the other components of the game react to the player decisions.

5.2.2.2. Movement Controller

The movement controller is only used to control the movement of the character, it receives inputs as a position or an enemy and makes the character move towards the point or enemy received, it also syncs the character animation so the character runs.

To implement the movement in an easy way it has been used the NavMesh system of Unity, this NavMesh system gets models set by the user as walkable terrain and creates a mesh that guides any object with a NavMeshAgent so it can navigate in the map.

The movement itself is done by the unity NavMeshAgent, which is an AI integrated with unity designed to move around NavMesh generated maps.

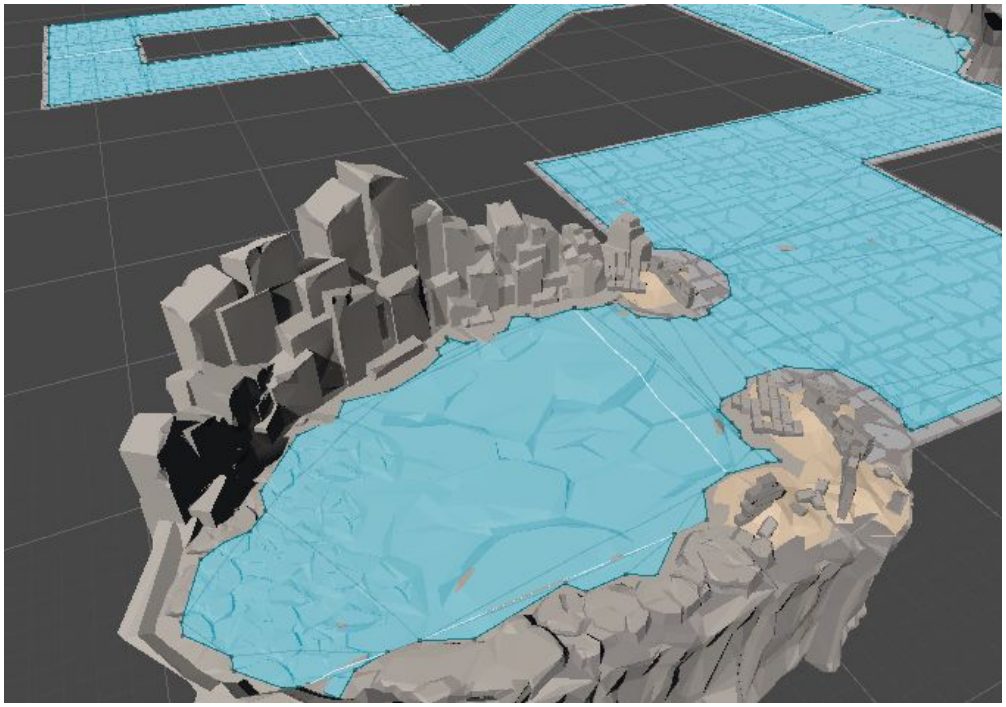


Figure 12: Navmesh example in the prototype level.

Unity NavMeshAgent handles the character movement by itself, as long as it's provided with a destination, the NavMeshAgent will search for the closest point to the destination in the Navmesh, and then move the character. So creating a the movement uning the NavMesh was just a matter of giving as destination the hit point of the raycast.

Apart from setting the destination it’s important to have an script that keeps checking the status of the NavMeshAgent in order to change animations of the character, and also be able to make other movements, such as keep updating the destination when the player move towards an enemy.

5.2.2.3. Skill Controller

The skill controller is the one in charge of casting and managing the skills the character has. It also stores all the possible skills of the character as well as the currently equipped ones, and allows the player to change them.

In order to organize the available skills and the equipped ones the skill controller holds two arrays of SkillData, SkillData will be explained in more detail later in the document but it can be understand as the skill itself for now. One of the arrays contains all the available skills for the character, and the other one has the skills that the player has equipped, so when the player changes the equipped skills the only thing needed to do is to swap the data from the equipped array with the other array.

To cast skills the skill controller receives calls from the character controller getting the hit point of the raycast and the playable character position, with these two points the skill controller computes the data to cast the skill, similar to the process done with the input, the skill controller must normalize the character position so it's at the same level as the enemy position. With both positions at the same level the skill controles can now compute the direction of the skill casted, this information will be stored inside a CastInfo class and will be sent to the skill casted.

One last thing the the skill controller handles is the skill casting limitations, which are attack speed, resource available and the skill cooldown.
To limit the mana cost the skill controller simply looks if the current resource of the player is higher than the needed to cast the skill, if it is enough it will be subtracted and the skill will be casted, if there isn't enough the skill won't be casted.

For the attack speed the skill controller uses a simple timer that diminishes each update and it's value is set every time a skill is casted, the skill controller will only allow a skill to be casted if the timer is below 0. This timer expresses the seconds or fractions of a second which are need to wait to match the attack speed values, for this prototype the attack speed value represents the number of attack per second that can be casted. It's possible to get the time needed to wait for the next attack by dividing 1 between the attack speed value.

The cooldown of the skill is mainly controlled by the skill itself, the controller is used to update the skillData cooldown, and as a way for the UI scripts to be able to access each skill and display the cooldown appropriately.

Figure 13: Skill cooldown display, skill 3 and RMB on cooldown.

5.2.3. Stats

The implementation of the character stats presented one of the biggest challenges that this project has, which was how to be able to link some kind of data, in this case a stat, for all the elements of project that will be using it, considering that some of them like buffs may be temporal. At the same time the stats also presented another challenge which was how to keep count of all the possible buffs that could modify this stats.

In order to overcome this first challenge of linking the stats data we used something similar to a singleton pattern using scriptable objects, the singleton pattern is roughly defined as having only one instance of a defined class so we can access it from other

parts of the code, which in this particular case would mean to have only one variable for the Max_Health stat.

However we don't want to have a pure singleton and create one class for every variable, but instead have unique instances of the same class for every stat, here is where scriptable objects come to help. Scriptable objects are an abstract class from unity from which you can derive your own classes, similar to the monoBehaviour class, scriptable objects are meant to be data holders, meaning that they lack most of the runtime internal calls of the game. Scriptable objects are used creating various instances of the class through unity and those instances are saved as .asset files, this functionality fits perfectly our objective of creating unique instances for every stat, all deriving from the same base stat class.

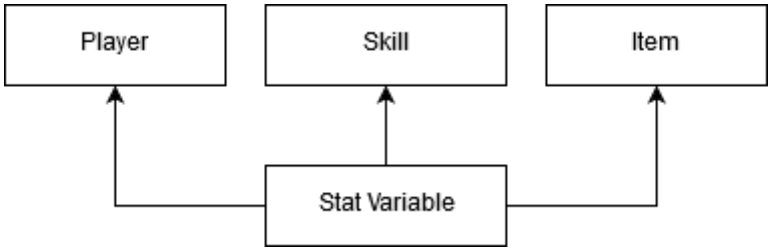


Figure 14: Simple diagram for the Stat Variable class

Now we can create a StatVariable class that derives from scriptableobject and then create different instances as .asset files, one for each stat we have, this instances can then be accessed by the various objects of our game. For the moment this stat variable will only hold a float that will be the value for the stat.

```
[CreateAssetMenu(menuName = "Chambers of Elrankat/StatVariable")]
[System.Serializable]
public class StatVariable : ScriptableObject
{
    [SerializeField]
    public float base_value;
}
```

Figure 15: Simple StatVariable

We still need to find a way to keep track of all the buffs that can be applied to every stat, in this case we will keep it simple and put 3 more float variables inside the StatVariable class, one for addition buffs, one for multiplication buffs, and one for the final buffed value of the stat, so we can use the value and the addition and multiplication value to calculate the final buffed value using the following formula: *buffed_value = (base_value + addition_value) * multiplication_value.*

```
[CreateAssetMenu(menuName = "Chambers of Elrankat/StatVariable")]
[System.Serializable]
public class StatVariable : ScriptableObject
{
    [SerializeField]
    private float base_value;
    [SerializeField]
    private float sum_value;
    [SerializeField]
    private float mult_value;
    [SerializeField]
    private float buffed_value;
```

Figure 16: Final StatVariable (float attributes)

Now the only thing remaining was to force an update of the buffed value every time the buffed value was used, at the same time we also protected the attributes and force to access the values via setters and getters methods in order to call a stat Changed event every time a variable is buffed, in case that some other parts of the game need to update some data based on some particular stat value.

```
    public float Base_value
    {
        get { return base_value; }
        set { base_value = value; stat_Change.Invoke(); }
    }
    public float Sum_value
    {
        get { return sum_value; }
        set { sum_value = value; stat_Change.Invoke(); }
    }
    public float Mult_value
    {
        get { return mult_value; }
        set { mult_value = value; stat_Change.Invoke(); }
    }
    public float Buffed_value
    {
        get { UpdateBuffedValue(); return buffed_value; }
    }

    void UpdateBuffedValue()
    {
        buffed_value = (base_value + sum_value) * mult_value;
    }
}
```

Figure 17: StatVariable value setters/getters methods

5.2.4. Skills:

The implementation of the skills was one of the biggest challenges of the project, the player could change the equipped skills at any time and so all skills had to share the same methods while they behaviour could be incredibly different. The main difficulty

though was that the skills had to be designed to be able to support changes in their behaviour or receive specific buffs to them, for example, a skill that shoots a fireball towards the cursor may need to have its behaviour changed to shoot 3 fireballs in an arc towards the cursor.

The first problem was quite easy to solve, as it only required a simple inheritance from a base skill class with various virtual functions so their sub classes could override them, this way each class will have their behaviour but all can be called using the same methods.

Making the skill able to support changes in their behaviour or buffs in their attributes however was a tough task, to start with it we took a similar approach to the StatVariables, all skills were into scriptable objects so buffs can be applied the same way its done the the stats, however this wasn't as simple as it seems, since scriptable objects are meant to be data holders they lack the functionality directly interact with objects from a game scene, meaning that it was needed to separate the skills in two parts, one the skill "brain" which is a scriptable object and holds all the skill data even how the skill must behave, and other part which is called skill instance, this second part inherits as a monobehaviour instead of a scriptable object and so it can interact easily with the scene, this second part is the "muscle" of the skill and will take all the data from the scriptable object and use it to make the skill appear into the game scene, one important note is that there could be multiple skill instance in the scene at the same time using the same skill scriptable object.

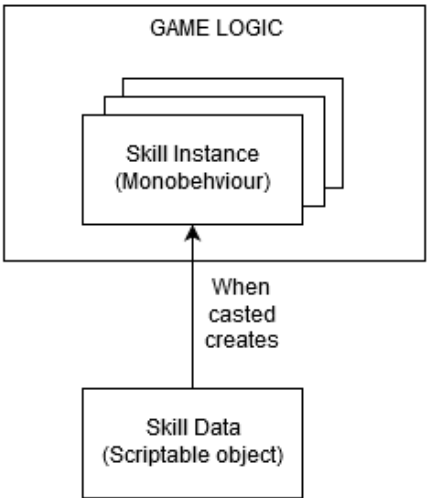


Figure 18: Skill Data-Instance relation.

For this solution to work it was necessary to redo the skill behaviour methods, since we needed something that one, had to be able to be modified and second it needs to receive individual information for each skill instance using them.
In order to make the skill behaviour able modified multi-delegates were used, delegates are in a simple way variables that store a reference to a function or method, which was helpful since it allowed to swap methods as if they were variables, multi-delegates expand this functionality allowing to store multiple methods into one delegate variable. This multi-delegates were used in the skill instance class in order to

set which method will it use to create the skill behaviour, and another multi-delegate was set into the skill Data class to be able to modify the cast method that will dictate how many instances it needs to spawn and also set the behaviour of those instances.

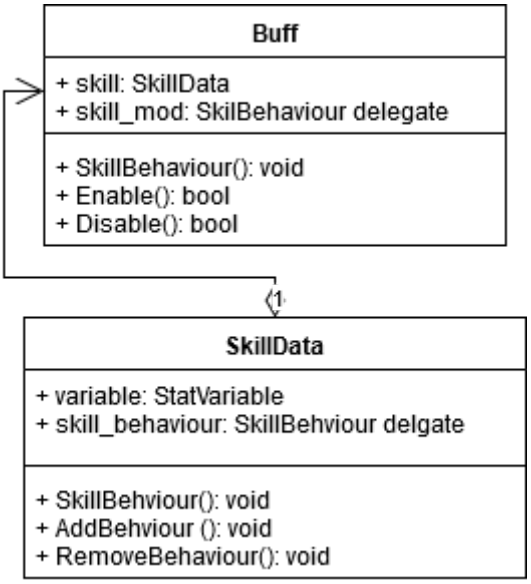


Figure 19: Simple SkillData UML showing multi delegate

Each individual instance needs to have specific data stored in them, like the time since they were created or the direction of the skill casted, this info can't be stored inside the SkillData class since SkillData is used by all instances of a skill, and changing values in it will change the behaviour of all those instances. In this case a CastInfo struct was created, this struct is stored inside every skill instance, this CastInfo stores the original cast data and the dynamic data for each skill instance, so it can be used by the behaviour method inside the SkillData scriptable object, this is needed since scriptable objects are saved as instances so changing the data of the SkillData class will affect all SkillInstances in the scene.

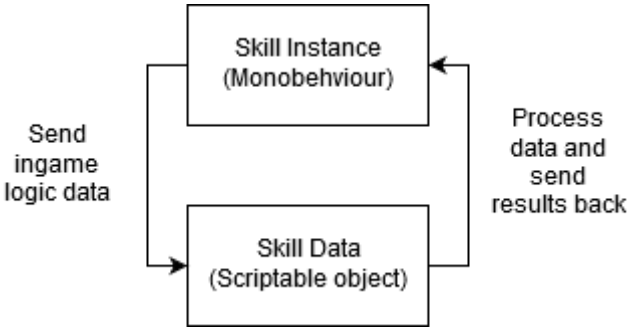


Figure 20: Skill Data-Instance update cycle

To recap all this explanation the skill code consists in two classes, one is the SkillData class, which can be considered the "brain" of the skill, this class stores all the data and the behaviour of the skill, then when the skill is casted this SkillData class created one or more copies of the SkillInstance class, the other skill class, this SkillInstances can be considered the "muscle" of the skill and execute the skill behaviour in the game world.

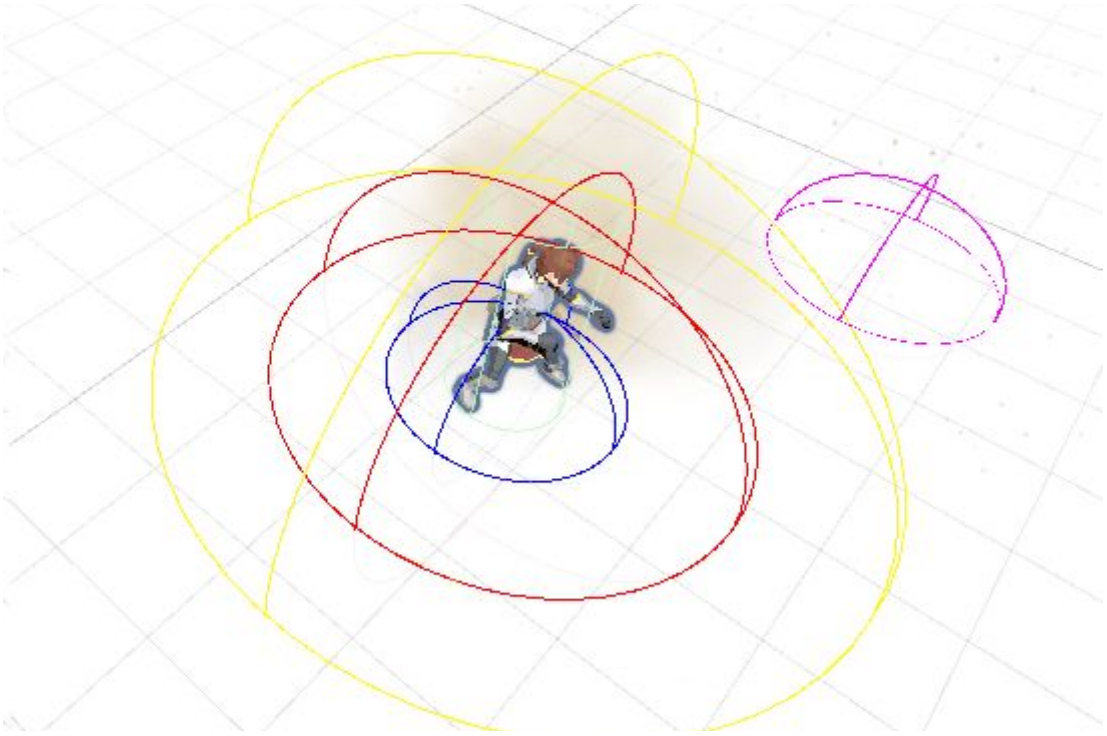


Figure 21: Skill ingame. Blue, red and yellow spheres represent skill reach, purple sphere skill objective (direction).

5.2.4.1. SkillData coding

To implement this concept the first step was to create the SkillData base class, first the SkillData class holds a few generic variables used for all the skills, such as the skill name or the skill icon. The most important part however is the 2 multi delegates mentioned before, one which hold the logic when the skill is casted and a second one that holds the logic when the skill is being updated in game logic. Important to remember that the first delegate will only be called once when the skill is casted, the second however one will be called each game update.

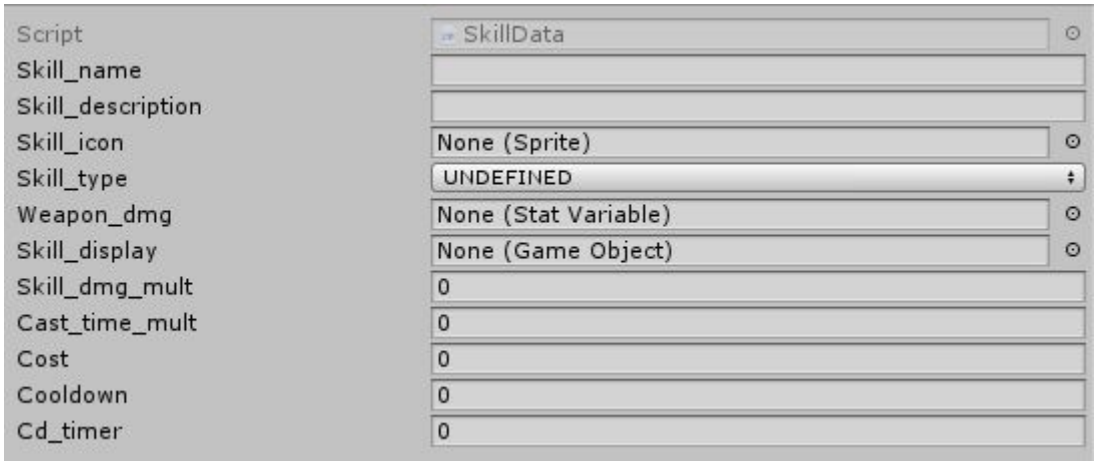


Figure 22: Base skill info (void)

Each skill have their own cast and behaviour methods defined individually, however the logic behind each skill is very similar in some cases, and can be divided in various archetypes which have the same logic behind them, and combined can give various skills as a result, these archetypes are explained below.

Skill Instance:

A common piece of code shared by all skill is the instance spawn, this part it's inside the cast method of every skill, and it's necessary for the skill to spawn their particles or object display inside the game world, and for this instance to know how to interact. This is done by instantiating a copy of the skill display data, and giving it the Skill instance component, one last step is need which is to initialize the Skill Instance, passing to it the skill behaviour method and the cast info for each particular skill instance.

Dash:

The dash archetype objective is to move the player from one point to another, jumping across map gaps if there is some. For this archetype it's needed a few additional data, the range, which tells how far the dash can be done, the precision which tells how much close to the objective point the player will be at the end of the dash, the speed which tells how fast the player will move during the dash, and a Game object which holds a trail effect that will be left behind, this effect is seperate from the typical skill display since it won't be used with the skill instance.

Regarding the logic of the archetype both the skill cast delegate and the skill behaviour delegate, the logic behind the skill cast delegate is very simple and it's used it's to prevent possible bugs, since we are forcefully moving the character having various dash behaviours at the same time may bug the character position and prevent it from moving, for this reason when a dash is casted it tries to register their game object id to the movement controller, if there is another dash behaviour registres the movement controller will fail the register, and the dash will be cancelled. If it can register to the movement controller the cast method spawns the dash trail particle and ends.

One last thing that the cast method does it to limit the range of the skill, if the skill receives a destination point more distant from the current position than the skill range, it replaces the destination point with a point at the skill max range in the same direction as the original point, which is achieved by multiplying the skill direction by the range of the skill and then adding it to the character position.

The rest of the logic is left to the skill behaviour method, to create the dash effect the behaviour waits for a few fractions of a second, 0.3s in this case and then teleports the player to the desired location, to simulate the effect of moving the behaviour moves the player towards the direction of the cast during the waiting period.

AoE checks:

To be able to hurt enemies or affect them with the skills it's necessary to check how many enemies or affected entities are inside the skill effect area, one common way to do it is by using collision system unity has, and check the collisioned object when we find one.

This approach however it's impossible to do with the system created, the collisions methods in unity are integrated with the monobehaviour classes, and in our system the logic is handled by the SkillData class, which inherits from scriptable object instead of monobehaviour.

In order to detect the other elements from the SkillData the collider search is done manually by using unity physics system collision search, with this it was possible to search all the colliders and their parent objects within certain areas, for this protype the areas used were a sphere, cone and cylinder.

The sphere and cylinder search are already done by the physics system of unity, so providing the sphere or cylinder dimensions it was possible to search all the object inside the sphere or cylinder, the cone detection however requires more coding, for this case the cone search was achieved by first making a sphere detection, and then checking the angle between the cone direction, and the direction vector from the center of the cone sphere to the detected entity, was smaller than the cone angle.

For single target skills the same collision check is made, then there is a distance check between all the colliders found and the origin point of the skill, then it's just a matter of discarding all colliders except the closest one the the origin point, which is the one that will receive the damage.

Multiple hits:

Creating skill which hitted multiple times was a matter of making an AoE check every update in the behaviour method, however some skills while they attempt to hit multiple times during their duration only wanted to affect the same enemy once, one example will be the earthquake, which leaves a zone that will damage and stun enemies that enter it but only once. For this purpose it was necessary to store all the hitted enemies.

Since it was impossible to store a list of hitted enemies inside the SkillData class, because it's shared by all possible instances of the skill, to solve this a collider list was added inside the ClassInfo struct, since this struct is saved inside the SkillInstance class it can hold all the collision of the skill.

Being able to hold the collisions found by the skill makes possible to check the previously hit colliders before applying damage, allowing to not apply damage to the already damaged enemies.

Projectile:

The projectile behaviour is entirely done in the skill behaviour method, and its objective is to make the a particular instance of a skill move in a particular direction. To have it the transformation of the skill instance was translated towards the direction of the skill by a magnitude marked by the projectile speed variable.

During the translation a spherical collision check is made to know if the projectile impacts with something, so we can act accordingly, at the same time it also checks if the projectile has surpassed its maximum range, when it does the projectile is destroyed.

Buff:

Skill that provide buff are quite different from the other skills, the biggest difference is that they only use the skill instance as a way of displaying particles so the player know they are active and update themselves, but their skill instance won't have any logic, and also contrary to the other skills they will only have on skill instance active at a time, and will do the logic from the SkillData itself.

For this skills the SkillData will hold the StatVariable which is being buffed, as well as the amount being buffed. Also due to the SkillData being the one which will operate all the logic this buff skill also hold a timer, and a boolean to know they duration and whether they are active or not.

Other than that the logic for these skills is pretty simple, with their cast method they will buff the StatVariable they are holding, mark themselves as activated and start the duration timer, in case they are already active they will simply reset their timer to extend the buff.

The only thing left to do is to deactivate the buff when the duration time ends, this part is left to the skill behaviour method which simply updated the timer, and, when the timer reaches the skill duration it takes of the buff from the StatVariable and destroys the instance so the particles are removed from the game.

Stuns, slows and push:

Stuns, slows and pushes are applied the same way as damage by calling function from the enemy script, however it's important to mention here because while they are not a logic part of the skills behaviour each skills that applies either a stun or slow to enemies must have defined in its SkillData the duration of the stun or slow, and in the case of being a slow also the magnitude of the slow, the same way when a skill pushes an enemy the skill must store the force of the push, which tells how far the enemy will be pushed.

5.2.5. Buffs

Buffs are a simple class whose only purpose is to modify to addition or multiplication modifiers of a StatVariable.

```
public class Buff
{
    public Buff(BuffType b_type, float b_magnitude, StatVariable b_variable)...

    public BuffType type;
    public float magnitude;
    public StatVariable variable;
    [SerializeField]
    private bool active;

    public virtual void EnableBuff()...

    public virtual void DisableBuff()...
}
```

Figure 23: Buff class

This class only has a few attributes to know which type of buff will be applying (addition or multiplication), the magnitude of the buff and a reference to the StatVariable which the buff will be applied.

```
public virtual void EnableBuff()
{
    if (variable == null || active == true)
        return;

    if(type == BuffType.BUFF_STAT_ADD)
    {
        variable.Sum_value += magnitude;
    }
    if(type == BuffType.BUFF_STAT_MULT)
    {
        variable.Mult_value *= magnitude;
    }
    active = true;
}
```

Figure 24: Buff enable method

The enable and disable methods first check if the buff is activated or not in order to prevent a buff applying more times than needed and then just modify the StatVariable values to reflect the buff.

5.2.5.1. Skill Buffs

There are special buffs that interact with skills, and are able to change their behaviour, either by changing how are they cast, or how they interact with the world after being casted.

While they inherit from the Buff class and use the same methods to Enable or Disable themselves, they don't affect any StatVariable, instead they modify a SkillData vairable, by adding one or more methods to their cast and skill behaviour delegates when they are activated, and remove then when the buff is deactivated. Below is a list of all the items with skill buffs in the prototype with an explanation of what they do.

Wind trident crown:

It changes the Wind Rush cast behaviour, makes the skill spawn 3 tornados instead of one, it's achieved by using the same method that casts the original tornado but rotating the skill direction.

Figure 25: Triple tornado spawn

Megumin Staff:

It increases the arcane explosion damage and explosion range, making a bigger explosion that deals more damage to more enemies.

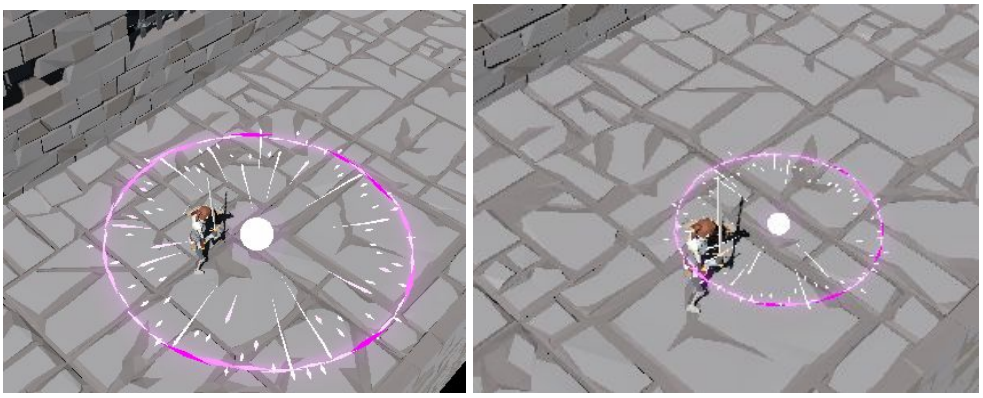


Figure 26: Arcane explosion size comparison

Gloves that let it go:

Doubles the area of the Ice Armor skill, also increases the slow magnitude to turn it into a root.

Synthetic Hydrophobic Invisible Tectonic Mace:

It affects the Thundercall skill, increasing it's cost by a huge amount but reducing it's cooldown to 0 and making it's area of effect bigger.

Figure 27: Thunder call size comparison

Vampiric grasp:

Changes the spirit refugee behaviour, damaging enemies inside it, and also reducing its current cooldown with each enemy hit.

Sacred arrow:

Makes the Wind Fury skill also increase the weapon damage stat.

Piercing needle:

Changes the Sting skill behaviour making it deal lower damage to all enemies hit, the first enemy hit still receives a huge amount of damage.

Dragon dancers:

Makes the Dragon breath skill damage enemies in a circle around you instead of a cone.

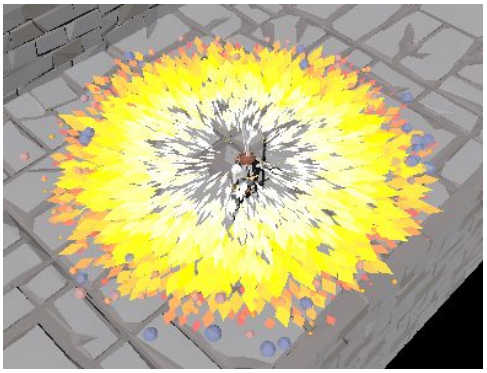


Figure 28: Dragon dance buff applied

3.14cm pants:

Changes the Wind rush skill behaviour, making the tornadoes spawned by it travel in a spiral by adding a circular motion to the original movement. It adds up with the wind trident buff, allowing to spawn 3 tornados moving in a spiral.

Figure 29: Triple tornado with spiral tornado combined

5.2.6. Items

Items themselves are basically data holders for buffs, so their are a simple class with some attributes for their basic data needed to interact with the other elements of the game, and an array of buffs. Then they had a function to activate or deactivate all their buffs.

The one who manages all the items the payer could have is the inventory class, this class has two arrays of items, one for the items in the inventory bag, so the player can collect and store items, and another for the equipped items. To manage this arrays the Inventory class has and Add/Remove item methods that first check if the operation can be done and then they execute it. The inventory class also has another two methods to equip and unequip the items, those methods are the ones that trigger the enable/disable methods from the items, so all their buffs activate when the item is equipped and deactivate when the item is unequipped.

The last part of the item implementations was the script for the item model in the game world, this script called ItemWorld, has a few attributes that indicate the radius in which the player needs to be to pick the item, the player position, and if the player wants to pick up the item. When the player clicks over an item the ItemWorld script sets itself to be picked and then starts to check the player position, when the player is inside the pickup range, it will try to add the item inside the Inventory, if it can it will destroy the item display in the game world.

5.2.7 Enemies

Looking at various games of this genre it was clear that enemies in this games are mostly just a way of obtaining various items, so their AI is really simple, it's only objective it to go towards the player and when it gets close enough attack it, there are some exceptions but their AI is still quite simple, like enemies staying at a fix position and throwing projectiles at the player or summoning more enemies.
For this game it was opted to go as well with a simple AI whose only purpose is to go towards the player and attack it when it is on range.

To achieve it the enemy AI has both a vision range attribute, an attack range attribute, and access to the player position, each enemy keeps checking if the player is or not inside their vision range, if the player enter this range the enemy will keep following them until it's inside their attack range, and then it will attack the player.

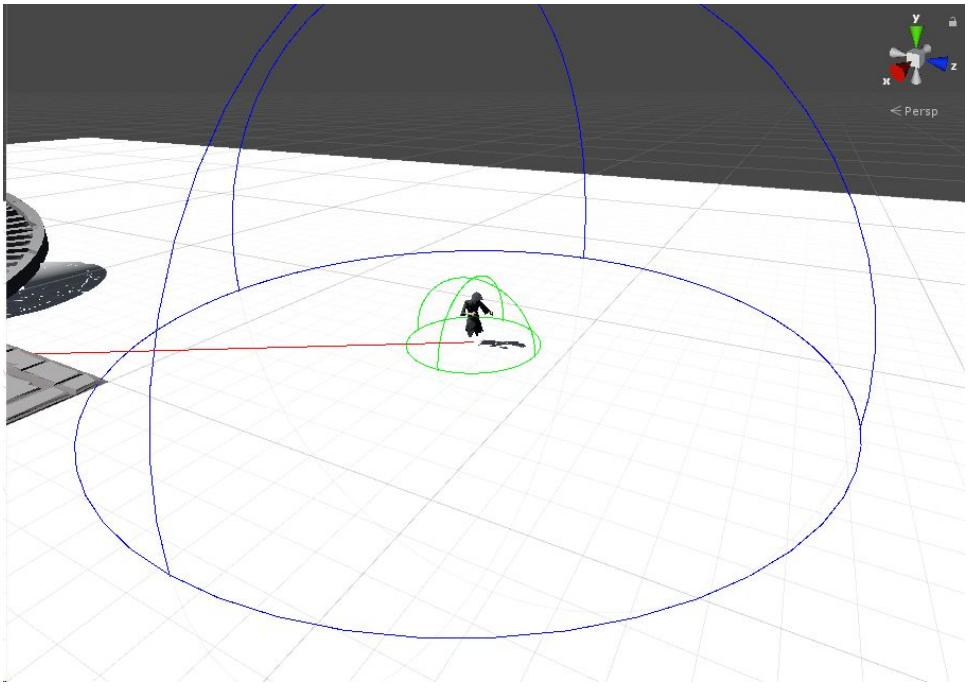


Figure 30: Enemy in the editor, view range in blue, attack range in green.

With this simple methods the basic enemy had a simple movement that was good enough to distract the player a little, however it was easily exploited if the player was at the limit of the vision range, to fix this, and also to make the behaviour a little less artificial the enemy was given a memory timer, so if the player leaves the vision range the enemies will keep following him until this memory timer runs out. Whit this little improvement the AI was not so exploitable and felt a little bit more natural when it was following the player, since it didn't stop instantly when the player left their vision range.

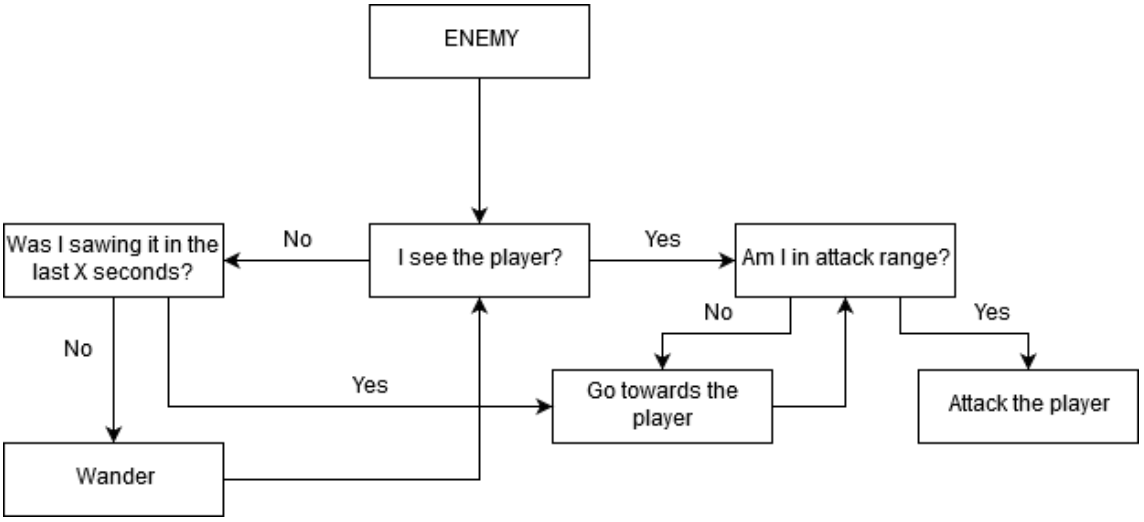


Figure 31: Enemy behaviour tree

Enemy AI also received a minor update to their behaviour when they were not following the player, instead of staying statics in a place enemies will randomly chose a spot around them to walk towards, with this enemies have a simple wandering behaviour that makes them feel a little bit more natural.

Enemies have different attack types, these types are related to the resistances that the player has, earth, fire, physic, shock, water. For example a player with high earth resistance and low shock resistance will take high damage from shock enemies, and little damage from earth enemies. To portray the damage type in the prototype enemies have a particle effect surrounding them that indicates the damage type depending on the particle color, green for earth, orange for fire, black for physical, purple for shock and cyan for water.

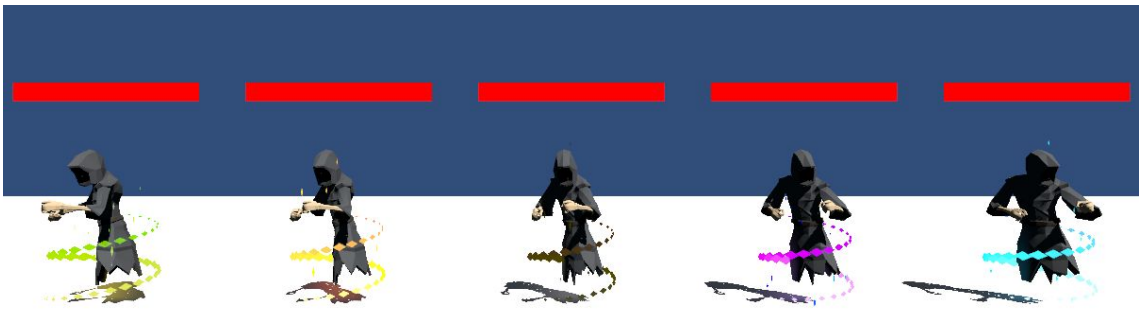


Figure 32: Enemy types, in order earth, fire, physic, shock, water

Another important thing to add to the enemies was some kind of feedback when the enemy has been damaged, so the player can know they are dealing damage. This feedback was first implemented using particle and sound effects, howevers with this type of game is typical to constantly damage multiple enemies at once, thus creating a

huge amount of particles and sound effects, that both lagged the game and also created a high sound disturbing the player. For this reasons the feedback was changed to a health bar above the enemy head that displayed their life percentile, which at the same time solved the problem of having the particles and sound and also allowed the player to see how much damage was dealt, making it easier for the player to compare skill damage.

6. Conclusions

During the state of art research it can be seen that this ARPG Diablo-like games can have very different approaches to their skill system, some games go for a complex approach with tons of decisions for the player to make in order to acquire skills, and some others gave the player an easier access to the skills just by playing the game. However there is one thing which all have in common, which is a complex and big item system that allows the player to customize his playstyle, either by changing their skills or their base stats.

Looking at the various approaches from other games it was decided to use a more simple approach handing all the available skill to the player from the start, this decision was made both to simplify the implementation in the game, and also because it gives the player more options to try out the different skills.

The itemization system was made but only to allow items to change player stats and skills, which made it really simple. The complexity of the item system in this types of games relies heavily on how the skills work, and also needs a huge amount of design work and analysis of various play testing session to adjust it accordingly. For this reasons the complete creation of this system was discarded, since it needed too much time to be completed and it wasn't possible. For this reason the implemented item system is a simple functional version, which changes skill behaviour and player stats, this is done only into a technical level, meaning that the progression of the character or the skill behaviour changes may not be correct, and even may be bad since all the design and testing needed for couldn't be done by a single person.

The approach in developing the skill system for the prototype has shifted a lot, especially at the beginning of the project where it was redone from scratch various times, mainly because the earlier versions weren't able to support the skill behaviour changes wanted for the prototypes, but another important reason was that the base skill class ended being way too abstract. This second problem was mainly due to a poor initial design, since the prototype was focused in the programing aspect of the game the initial design was very simple, and only defined a few skill types, such as having a projectile, or having a damaging zone.

This vague design ended in the creation of base SkillData class being too abstract, since it tried to allow for a wide range of behaviours that weren't planned for the prototype, and it was needed to define more aspects of the skill within the inherited classes, while this seemed good at creating new skill, it was too complicated to change their behaviour with items or interact with them, because a big part of their behaviour was inside the inherited class, making it almost impossible to use the virtual functions from the base class to modify it, which defeated the purpose of having the base class.

After making a more concrete design of the game made more clear the limitations of the skills, by having a more clear idea of the not needed behaviours, it was easier to limit the functionality of the base skill class, which allowed to create a base skill class

that was concrete enough to be used by its own, and abstract enough to change its behaviors accordingly.

Looking at the current state of the skill system and seeing how it's able to change the behaviour of the skills any way we want, by adding more methods which change the logic of the SkillData class, there is a clear upgrade which can be done to the system, which consists on splitting the different skill archetype behaviors from the SkillData class itself, and keeping the SkillData class as a container of the skill archetype behaviors, allowing to create new skills by combining the different archetypes, creating a modular skill creation system.

This will be powerful tool for designers to easily create and test new possible skills, since they will only need to create a new SkillData instance and add the behaviours they want to create new skills. But it will be possible to expand even more, since the skill change behaviour can happen in real time within the game, it will be possible to handle the players all this options and allow them to collect various skill archetypes, and make them create the skill they want to play with.

7. Bibliography

- List of Game Engines: https://en.wikipedia.org/wiki/List_of_game_engines
  [Last consulted on June 24th 2019]
- PoE skill introduction: https://www.youtube.com/watch?v=0dVirawNjwM
  [Last consulted on June 24th 2019]
- Agile methodologies:
  https://resources.collab.net/agile-101/agile-methodologies
  [Last consulted on June 24th 2019]
- FDD: https://en.wikipedia.org/wiki/Feature-driven_development
  [Last consulted on June 24th 2019]
- Unity: https://unity.com/
  [Last consulted on June 24th 2019]
- Unreal: https://www.unrealengine.com
  [Last consulted on June 24th 2019]
- Scriptable objects arquitecture:
  https://www.youtube.com/watch?v=raQ3iHhE_Kk
  [Last consulted on June 24th 2019]
- Robert Nystrom, 2014. *Game Programming patterns.*
  Web version: https://gameprogrammingpatterns.com/
  [Last consulted on June 24th 2019]

8. Annexes

- **Github:** https://github.com/Yulna/TFG_ARPG-Game

  Github repository for the project files, external assets not done during this project are not included

- **Release:** https://github.com/Yulna/TFG_ARPG-Game/releases/tag/v1.0

  Release page on github of the final build